# Gratitude++ Final Report

Joel Scott & Benjamin Seifried

December 12, 2023

[Gratitude++ Final Report 1](#_Toc863148778)

[Introduction 1](#_Toc983314113)

[Technology 2](#_Toc463891985)

[ReactJS 2](#_Toc1381837141)

[Node.js 2](#_Toc138661887)

[CSS Stylesheets 2](#_Toc461082695)

[Firebase Real-time Database 2](#_Toc496367264)

[GitHub Version Control 2](#_Toc22323054)

[Replit.co 3](#_Toc1294607642)

[Development Experience 3](#_Toc5282199)

[Design 4](#_Toc504389318)

[Database 4](#_Toc1017415679)

[Login 5](#_Toc244917392)

[Homepage 6](#_Toc1887158252)

[Journal Entry Creation 7](#_Toc1106796415)

[Homepage journal entry view 8](#_Toc1314716276)

[History 9](#_Toc1574993893)

[Community 10](#_Toc259179170)

[Resources 11](#_Toc681099081)

[About 11](#_Toc1627767612)

[Settings 11](#_Toc1296664735)

[Logout 12](#_Toc1872509989)

[How to deploy the project 13](#_Toc1596975140)

[Known Bugs 15](#_Toc1155018034)

[Future work 16](#_Toc1828982188)

[Expanded Word Cloud Filtering 16](#_Toc2110248447)

[Theme Selection Component 17](#_Toc722806679)

[Expanded Administrator Functions 17](#_Toc1331326524)

[Cloud Masking Element 17](#_Toc249163263)

[Audio/Video Content Expansion 17](#_Toc534572459)

[Keeping user data 17](#_Toc680477520)

[References 17](#_Toc471831894)

## **Introduction**

The purpose of the project that we have created throughout this semester is to create a website that puts forth the idea of gratitude. What this means is that the website aims to help people by creating an area where they can put down personal thoughts and feelings that will allow them to express themselves.

In the world today there seems to be a negative energy that surrounds everything, and the goal of Gratitude++ is to fight back against these negative ideas and feelings and promote positivity and mental wellbeing. The problem that this site aims to solve is to get people away from these negative influences in their everyday lives such as social media. Social media algorithms nowadays find that they can get more people through curating and prioritizing negativity over positivity.

Using various methods described in this report, the Gratitude ++ Journal Application aims to help focus positivity and increase a user's sense of wellbeing.

## **Technology**

The technology that was used to build the website was a mixture of React.js, Node.js, CSS stylesheets, and Google Firebase for the creation of the project, GitHub for the version control, and Replit.co as the server from which to host the web application for the prototype deployment.

### **ReactJS**

React.js is an open-source JavaScript library created for building user interfaces, particularly for web applications. React allowed us to create a Dynamic User Interface and reusable components in what is considered a component-based architecture.

Our React based application uses a virtual DOM (Documented Object Model) to optimize performance. Instead of directly interacting with the browser's DOM, React creates a lightweight copy of it in memory (virtual DOM) and updates only the necessary parts when there are changes.

Most of the files created in the application used the JSX syntax extension provided by React that allowed us to write HTML-like code directly in JavaScript. These elements were then routed through the app.js file which was in term rendered by the index.js file.

The React Hooks feature allowed us to use state and other React features in functional components. They enable the reuse of stateful logic across components. [1]

### **Node.js**

Node.js was used to run JavaScript code on the server and enabling the execution of JavaScript outside the browser. It provided a means of working between the front and the back-end Firebase Real-Time Database. [2]

### **CSS Stylesheets**

Most of the styling was accomplished using pure CSS rather than libraries. The Word Cloud Component was initialized using the react-wordcloud library but was modified using CSS as well.

### **Firebase Real-time Database**

Firebase Realtime Database is a cloud-hosted NoSQL database offered by Google as part of the Firebase platform. Because it is a NoSQL cloud database, we adapted to storing data in JSON format. Google Firebase was also used for user authentication and control. [3]

### **GitHub Version Control**

All version control and documentation were passed through a Github repository accessible to both developers and all stakeholders. Both developers were given admin privileges to the repo so that they had unfettered access to push, pull and merge without restriction. [4]

### **Replit.co**

The application was hosted by Replit.co which allowed us to synchronize our changes with the connected GitHub repository. The free version of the virtual CPU became inadequate as the code base grew. Both developers opted to upgrade to the first tier of the paid version to avoid performance bottlenecks. [5]

### **Development Experience**

This was the first time that either developer used ReactJS, and the learning curve was relatively steep. Many hours were spent looking over available documentation and watching videos to acclimate and familiarize ourselves with the technology. As development progressed the use of the React DOM and React hooks became increasingly intuitive.

Towards the end of the development, it was understood why many in the field have criticized certain aspects of the technology. While extremely useful in server-side development and in creating and designing a user interface, the React architecture seemed to expand and bloat when elements such as confirmation modals and message dialogues were introduced.

What at first seemed to be a lightweight tool that allowed modular development soon seemed to become a bloated code base that needed detailed commenting and documentation to ensure that the code was easily readable and comprehendible. In future development projects we believe that the exploration of alternative frameworks would be ideal. Perhaps VueJS or AngularJS.

The use of Node.js was intuitive once the developers both came up to speed on the use of JavaScript and the Firebase Documentation for making queries to the Real-time Database.

The Firebase Real-time Database featured a NoSQL database structure. This was the first time either developer used this kind of database structure. The learning curve required a similar amount of preparation to the integration of ReactJS. Many hours were spent pouring over documentation and video tutorials. In the end, the fact that the data structure relied on JSON style file structures made the creation and implementation accessible.

The use of Github for the repository and version control was familiar to both developers, though not at the scale to which it was used in this project. Eventually a set of bet practices were agreed upon that made the version control easy and accessible to the team without the need for a singular administrator approving pull requests.

The choice of Replit.co to host the prototype of the application was an appreciated suggestion and has worked out well for the team. This was the first time either developer had explored the use of replit.co beyond the previous uses as a sandbox development environment. After a bit of exploration and reading of documentation, the team is happy with the ease of use and adoption.

## **Design**

### **Database**

Gratitude++ uses Firebase Realtime Database as the storage system for all relevant data. Realtime Database is a no SQL database whose format resembles more of a JSON given format rather than a standard table. Inside the database it is broken into three primary nodes to hold all the user’s information for the site. The first node that it is split into is the “users” node, which is where all information relating to each specific user is stored. Under this node users are defined by their UID, which is a unique identifier given by Firebase through their Google login procedure. Due to the nature of how the UIDs are created, they ensure that no two users will have the same UID, making it easier to create a location for that specific user. After a UID is created for a specific user, they will keep that same UID every time they login through the Firebase authentication services, allowing for their data to be retrieved by using this information to find their specific location.

Within the user’s node information based on their account is stored, such as their username, their profile picture, the list of posts that they have created, and a Boolean variable to determine if they are an admin or not. Under the post node relevant information for the post is created, such as the content, the headline, the date, and visibility for each post. The visibility will determine if the post will also be pushed into the community node as well if the visibility is set to true. To create the list of posts with unique identifiers for every entry, the Firebase push function is used, which pushes items into a specific location with a new unique identifier as the parent node to the data.

The next node is the community node, which is responsible for holding all information relating to the community page. This node is amended to every time a user creates a post and sets the visibility to true, allowing them to push things into the community for all to see. Under the post node exists each post for that have been pushed in chronological order, which is the way that it is done through the push method mentioned above in the user’s node. Within each post child node there is the headline of the original entry, the date it was created, the username of the original author, a list of all users who have flagged the posts, and the number of times the given post has been flagged. There is currently no direct way to remove data from the community node without the user deleting the information from their account, which will also remove it from the community node. The community node does not house the content for the user’s entries, it instead holds information so that the data may be retrieved from under the user's node. This was done to avoid having to double down on writing the data into the community node and the users node.

The final node that exists to house information is the username node. For the purposes of security, this node was created to hold a list of all existing usernames for the purpose of unique usernames during account creation. This node has a listing of all currently created users by their username and is only used when users attempt to create a new account. A call will be made to this node when users create accounts, and it will determine if the username is already taken and return either true or false. If it returns true, the username that the user has just chosen will be added to the username's node.

### **Login**

The login component begins by importing necessary styles, React, and several functions from other sources. Notable functions that are imported are signIn, userAccountCheck, getUsername, and getUserImage, which are essential pieces for the overall system that contribute to user authentication and account management. The auth variable from the ‘firebaseConfig’ file is also imported to interact with authentication services.

The login component uses the ‘useEffect’ hook from react to subscribe to changes in the authentication state of the user. This allows the system to dynamically respond to changes within the sign in state and handle changes if the user’s authentication state is updated. Upon the update of the user authentication state, a series of actions are taken, such as calling userAccountCheck to determine if the user has an existing account and setting the user’s uid into the local storage for use.

When a user is authenticated, the component will check if the user has an existing account or not. This is done by calling userAccountCheck, which returns either true or false depending on the state of the user’s account. If the user has an existing account, getUsername and getUserImage will be called to get information from the database. This information will be stored in the local storage for use on other pages, such as the homepage to display the user’s account information at the top of the screen. If the user does not have an existing account, they will be navigated to the newUser page, where they will create a new account.

The signIn button within the react component will call the signIn function, which handles the user sign in procedure through firebase. Upon clicking the button, users will be prompted to enter their email address and password to login through Google. The signIn function calls Firebase’s signInWithRedirect method, which signs the user in by redirecting them to a separate page to handle the sign in procedure. Because Firebase is handling the sign in procedure, the authentication checks are done through their services to determine if they have a valid token from Google to access the site. All errors that occur through the redirected sign in process will be handled by Firebase before the user returns to the login page.

Users on the login page will be required to be authenticated in order to access the site. By using the react routing system, all pages are accessible by typing /pageName, where pageName is the name of the given page in the routing. The website will use the aforementioned authentication status of the users to grant them access to the site. If the users do not have valid authentication through Google services, any attempt to access the site through the url will return them to the login page, forcing them to sign in to access features of the site.

### **Homepage**

After the users have logged in and have a valid account, they will be brought to their personal homepage. The homepage, also known as home, is a page that is made up of a collection of three different components. The page is designed like this to allow the components to render independently of each other, allowing for a quicker display of the homepage. The page imports the necessary CSS for the basic styling of the site, and imports wordCloud, TopBar, and homeArticleSearch. In the page a button is also created that will take the user to the journalEntryCreation page.

Looking at the wordCloud component, it begins by importing all necessary styling from the CSS file and imports functions that are necessary for fetching data that will build the word cloud. These functions include searchJournalEntry and wordCloudList. The component also imports react word cloud, whose features are used to help build the word cloud with the provided data.

The word cloud is designed by using an asynchronous function fetchData, which is responsible for gathering all the necessary data that will be used inside the word cloud. Due to the nature of Firebase functions being asynchronous, creating the async function allows for Firebase functions to be called with the await keyword, such as

* const wordCloudArray = await wordCloudList();

Doing this forces all code following the await keyword to pause and wait for the action to be completed, which in the above case is calling wordCloudList. Creating a variable for the function will store the returned value from the function in the variable, allowing for it to be used later in the component. The function wordCloudList makes a call to the database, where it gathers a list of all the user’s headlines from entries they have created and compiles them into an array. In the component multiple filtering methods are done to shrink the size of array that will be used to create the word cloud. Stop words, commonly encountered and non-informative words, are excluded. Duplicate words are eradicated, and a diverse set of unique words is curated. The result is a refined list of words, each assigned a randomized value for size in the word cloud. A random number generator is used to generate values ranging from 50 to 100 in increments of five to determine the size of the words in the word cloud. Once the words are filtered into an array, they are randomly shuffled within the array. The first fifty or less values are then taken to create the word cloud.

Once the array for the word cloud has been created, it will be displayed inside of the wordCloud component. Utilizing the ReactWordCloud library features, the word cloud will be generated in varying colors and sizes based on the words in the given array. Users will be presented with an immersive experience as each word in the word cloud will also be clickable. Clicking on any word within the cloud will show the user all entries that contain that given word, which is done by getting the word through the react library features and calling searchJournalEntry with the given word.

The next component that makes up home is the homeArticleSearch component. This component is primarily used to render a list of entries created by the user in order of newest to oldest. The component begins by importing all the CSS and imports the homeJournalList function, which is used to create the list of entries, and React Router Link, which is used to link to other pages using react’s routing process. Beyond just the basic list of articles this component also features pagination for the articles, allowing the component to render five articles at a time and have page numbers and buttons at the bottom of the page used to scroll through the listing of articles.

The main feature of the component is the rendering of the articles, which is done by getting a collection of all articles from the user’s database node to populate the component. Utilizing the homeJournalList function from the databaseHomepage module, a list of articles associated with the user are collected and rendered onto the page. The rendering is done by using features of react, the react useState hook, which allows for data to be updated dynamically and render the data when it is available. Once the function has finished being called, the result will be an array consisting of all values in a given format for display. The list of artricles is then reversed to put it order from newest to oldest before displaying the articles.

Each of the entries that will be displayed will be clickable, allowing the user to see the content of the entry they have chosen. This is done by passing the information for the entry as a string and using the title and date to find a given entry. The entry string is put into the local storage, where it can be retrieved on another page where the display function is called to show the content. This is accomplished by using the map and Link features of react, which allows for the entry to be displayed and get the result of the clicked value and allows clicking on the entry to link to another page using the react routing.

### **Journal Entry Creation**

Journal entry creation is at the forefront of what is being done with Gratitude++. With the site being a journal entry website, journal entry creation is one of the most prominent features of the site. Using just a couple of input boxes, a toggle switch, and a submit button, this feature is simple. Unlike many other pages that make up the site, the journal entry creation is a single page rather than a collection of components. This is done because there is no rendering necessary on the page with it being static in nature, so the only changes that will occur will be a direct result of the user’s actions. Beginning with the imports of functions, styling, and libraries, the journalEntryCreation page establishes a foundation for a user-friendly experience interface. It integrates various components, such as “ToggleSwitch,” “JournalPrompts,” “Confirmation Modal,” “TopBar,” and “MessageDialog,” each contributing to specific functionalities within the journal entry creation process.

The journal entry utilizes simple features such as trim and a regular expression to help with the checking for the entries. Each entry must be a minimum of five characters, which is checked after all additional whitespace is removed from the string. If the entry meets the character minimum it is checked against the maximum, which is 100 for the headline and 1000 for the content. Message dialogs will appear telling the user of any information that does not meet the requirements, and it will kick back the submission. If the entry meets all the criteria a function createNewEntry() is called, which uses the provided information to set the information in the database. Using features of firebase each new entry is given a randomly generated key that marks the location of where it is stored to make sure that no data is overwritten. Once the data is placed a message box will appear alerting the user of the successfully completed task, and then it will navigate the user back to the homepage.

Within the journal entry creation there is also a call to the function wordCloudList(). Calling this function gets the latest journal entry created by the user and puts it in the local storage to be used on the homepage. This is done asynchronously to ensure that the actions are completed before returning the user to their homepage. The purpose of generating the array on this page is to lessen the time required for the word cloud component to render when the user navigates to their homepage. By updating the information in the array as the information becomes available it is available for use as soon as the user returns to their homepage to reorganize and display.

### **Homepage journal entry view**

The journal entry view page is a major part of the overall site. It is where users will have the ability to look back at previous posts they have created and delete the existing posts if they choose to do so. The page is created by importing the necessary CSS, the TopBar component, and several functions designed to help with the display of the entries and the deletion of the entries. Other things that are imported include react dependencies, which are essential for doing this like dynamically updating the information and managing the dialog message that appears upon user actions.

The page begins by creating a list of variables for use and gathers a string from the local storage to use in the retrieval of data. By calling some imported functions, such as entryHeadline() and entryDate(), the string can be broken apart and the relevant information can be acquired. Using the headline and date, a function called getUserEntryContent is put inside the use effect, which allows it to be called asynchronously and wait for the data to become available before running the function. This function goes to the database and finds the headline and date that match the parameters, and then returns the content through the function to be displayed on the page. If the content cannot be found a message saying no content will be put in its place and an error will be handled.

Beyond just the viewing of entries, users will also be given the option to delete the entry if they choose by clicking on the trash can icon on the homepage journal entry view page. When the user clicks the icon, it calls the deleteJournalEntry function, which takes in the headline and date as parameters and deletes the entrie entry from the datbase. This function is run asynchronously, so it waits for the operation to complete before it navigates the user back to their home page. If the entry is on the community page, meaning that the visibility of the entry is set to true in the database, then the entry will also be deleted from the community page. This is done by using the entry and date and username of the user to find the entry in the community node and delete it from the community page. This is done inside of the deleteJournalEntry function by calling a separate function to run asynchronously before the information is deleted from the user’s node to ensure that the visibility is true before deleting.

As with all submit buttons within the site, there will be a confirmation modal that will be opened when the user clicks the icon. The user will be given two choices in the modal, to confirm the selection or to cancel. If the user chooses to confirm, the above functions will take place, and the user will be given a message to alert them that the entry has been deleted. Using react router the use will then be redirected back to their homepage and the word cloud information will be updated to remove information from the entry that was just deleted. If the user clicks cancel the confirmation modal will close and no changes will be made to the user or the entry.

### **History**

The history page is where all past entries for the user will be displayed similarly to how it is done on the homepage. Unlike some other pages, no functions are imported to this page, but rather a list of libraries and components from elsewhere in the project to create the page. The first component that is imported is the history search bar, which is used to allow the user to search through their list of entries rather than having to scroll through them all. The second component that is used is the HomeArticlesSearch. This component is the same as the one that is present on the user's homepage, keeping with the pagination and the display of only five articles at a time. The component is the same to keep with the style and format of what currently exists, with the only major change being the addition of the search bar for the users.

The search bar component begins by importing the CSS, the react libraries, and a function called searchJournalEntry from databaseHomepage. The searchJournalEntry function is used to search through all entries in the user’s specific node and return a complete list of all items that match the search, or a message that tells the user that the result could not be found. Using the user’s input as the parameter for the search, the function converts the input to lowercase, as it does with all the headlines that it searches against and puts any found items into an array. Once all entries have been looked through, the function will return the array to the history search bar component. The results are put into the local storage, where it navigates to another page specifically designed for displaying the results from the search, called searchResults. On the searchResults page the user may click on any of the entries that were found to see the content of the entry, or they may click the back button whick will take them back to the history page. If no results are found through the search, the user will still be taken to the searchResults page, but a message telling them that no results were found will be in place of where the entries would be.

### **Community**

One of the major focuses of the site is the community page, an area where users will be able to share the entries that they would like others to be able to see. The community page is split into three separate components, the community word cloud, the community search bar, and the community entries. Each component helps to make the community feel like a welcome and inviting place for all users and provides a place where people may look upon posts that others have created if they choose to do so. Many of the components that make up this page follow a similar pattern to how others throughout the site are done, with a few key differences to make it unique and to display the information in the best way possible.

The word cloud for the community page is done in a very similar way to how it is done on the homepage, with a few slight changes to ensure that it displays better for the community. The general construction of the cloud is the same, but the location of the data is different from that on the homepage. In the community cloud data is collected from the community node to present data from all entries that users have marked as public. A major difference is the way that the cloud is displayed because on the community page, the word cloud will generate an array of values to populate the cloud. Once the array has been created, the values are then randomly shuffled in the array and the first fifty values are taken to create the word cloud. Doing this ensures that data from all the community page will be present when viewing the word cloud, rather than data from the most recent posts.

The next component on the community page is the community search bar. Similar to how the history search bar is done, the community search bar takes in the user input and searches through a list of posts to find all matching entries and display them. The major difference between the two searches is the added feature on the community page to search by username. Rather than just displaying a list of all entries that match the given input, the community page also displays any usernames that match the input. This allows users to search by username and by headline, giving them added functionality on the page and allows them to do a deeper search. Due to how the database is set up, this process just does an additional check on the data to see if it matches the headline or if it matches the username and follows the same practices to display all information retrieved from the search.

The final component on the community page is the community article search, or the display for the community page. The way in which the data is displayed is the same as it is on the homepage, but the overall information that is displayed is slightly different. Clicking on any entry will link to another page where the username, data, headline, and content is displayed. This is done by retrieving all that information from the database under the node of that specific post, which is confirmed by checking the headline, username, and date to ensure that all the values match. An added feature to the community page display is the ability to flag a given post on the page. Using features of react, a small flag icon is put onto the page with the entry the user is viewing. When a user clicks on the flag, a check is performed by calling a function checkExclusionList(), which calls to the node where that post is and checks to see if the users are on the exclusion list. The exclusion list is a listing of all users who have previously flagged this post, and by checking the list and returning a Boolean based on the result, it ensures that posts cannot be flagged multiple times by the same user. If the function returns true, a message is displayed to the user that they have already flagged the post. If the function returns false, the number of flags to that entry is incremented by one, the user is added to the exclusion list, and the number of flags currently on the entry is returned. If the number of flags is greater than or equal to five, the post will be removed from the community page by calling deleteFromCommunity, but this will not affect the entry on the user’s personal page. This feature allows users to flag posts they find offensive, but the current number of flags for any given post will not be shown to avoid having users band together to flag every post.

### **Resources**

The resources page is very simple and straightforward and requires the least amount of code to make it work. There is no connection to the database on this page and no additional functions are imported to make it work. All this page consists of is a collection of links to external sites which is done by using simple html and CSS to display the information. All the links are put into an array, and those links are then displayed using the <a> tag through html to create the hyperlinks to the external sites.

### **About**

The about page for the site is the most basic of pages on the site. Its only goal is to provide information to the user to help them understand how the site works. The page is done almost entirely in html, without needing to use any functions or additional react code to make it work. The page just uses headers through html to create sections that give a brief description of each page and imports the react router library to add the back button onto the page.

### **Settings**

The settings page uses several components to create it, but most of them are rendered conditionally. Starting with the file, the CSS and necessary react libraries are imported, as well as a few other functions necessary to work with the user settings. Several modals and react routing are also imported to make the user settings page work. The page itself serves four distinct purposes, the creation of user accounts, the updating of user profile pictures, the deletion of user accounts, and the admin functionality. The page was created in such a way that additional pages were not necessary for all the features, and using react certain components are only displayed if certain criteria are met.

Beginning with the user account creation, this feature adds an additional box onto the account creation display to allow users to enter their username. The rendering of this depends on the user's state when they first log into the site. When the user logs in, a variable will be set in the local storage called isUser that will evaluate to true if the user has an account and false if the user does not. This is done by checking the database to see if there is an existing UID to the one the user was just given through the Google login. If the user’s isUser value is false, then the newUser component will render the box to enter a username as well as selecting a profile picture. When the user clicks submit to create an account, a function usernameCheck will be called, which will check the username against existing ones to determine if the username is taken or not. The usernames will all be made lowercase to avoid matching username, even with different casing. Beyond the checks for duplicate usernames, usernames must also be a minimum of five characters, must not contain spaces, and must be shorter than 20 characters. If the username meets all the criteria, their username is then added to the username node, and a node based on their UID is created that holds their username, their profile picture, and their admin status. If the user already exists, the input box will not appear and users will only be able to update their profile picture, which will not affect any other data in their node.

The next area for users is the ability to delete their accounts. If the user chooses to delete their account, they will need to click on the delete account button on the user settings page. This button will open a confirmation modal asking for confirmation for this action. If the user hits confirm, a function called deleteEntriesFromCommunity() will be called, which will delete any entry that the current user has created from the community page. This is done asynchronously to ensure that the username is not wiped from the local storage before the process is completed. After the function has finished, another function called deleteUserAccount() is called, which will delete the user’s node based on their UID, subsequently deleting all child information as well. The user’s username will also be deleted from the usernames node so that other may use the name. After the account has successfully deleted the user will be given a message telling them that the process was successful, then their local storage will be cleared and using the react router they will be moved back to the login page.

The final component in the settings page is the admin component. This component allows for any user who is identified as an admin to delete a given user account. This component is rendered conditionally based on the user's status as an admin. When they navigate to the settings page, a function called admin() is called, which returns true if the user is an admin and false if they are not. A user is currently only made an admin by directly updating their node in the database and setting isAdmin to true. If they are an admin, they will be shown a list of all currently existing usernames with a delete button next to each one. If the user clicks to delete an account, a confirmation modal will appear asking the user to confirm their choice. Upon clicking to delete, a very similar process to the one described above will take place. The only difference between the user account delete and the admin account delete is that the admin will not be moved to the login page after the account is deleted.

### **Logout**

Logging out of the site is a simple process as it uses firebase functions to log the user out. Upon clicking logout, a call to the firebase function signOut is made, which takes in the user’s authentication variable as a parameter to log the user out. After the user has successfully logged out, their local storage and session storage will be cleared. This is done to keep with the sites security and prevent users who are not logged in from accessing the site.

## **How to deploy the project**

To deploy the project, an active Google account and an account on Replit will be required. The first step of deploying the project after getting the GitHub repository is to go navigate to Replit.com to begin creating the server for the project. Once logged into GitHub, click on create repl and click Import from GitHub. Take the url of the GitHub repository and paste it into the box under from URL, then click import from GitHub. Once the repository has been imported, the site will navigate to the page for the project and will create a console and a shell. The next step is to click on the shell and run the command “npm install –legacy-peer-deps" to install necessary packages for the project to run.

After all the required packages are installed, the next step is to go to firebase.google.com to configure the database for the project. Once at the site login to firebase through google and navigate to the firebase dashboard to see a list of current firebase projects. Click on add project and follow the on-screen instructions to create the firebase project, disabling google analytics. After all the information has been entered click on create project, where it should redirect to the dashboard of the newly created project. In the dashboard click build and click on Realtime Database. In the console click the shortcut for Realtime Database and click on create database. For the security rules of the database start in locked mode and create the database. Once the database has been created, copy the following security rules and paste them in place of the existing security rules in the database:

{

"rules": {

".read": "auth != null",

".write": "auth != null",

"users": {

"$userId": {

".read": "auth.uid == $userId",

".write": "auth.uid == $userId",

"posts": {

".indexOn": ["Headline"]

}

},

"usernames": {

".read": "auth != null",

".write": "auth != null",

}

},

"community": {

".read": "auth != null",

".write": "auth != null"

}

}

}

The next step is to click on project overview in the firebase console and click web in the center of the screen. Under web give the application a nickname, ignore the checkbox for enabling firebase hosting, and click on register app. On the next page copy the firebase config information from the use npm section and click continue to console. Return to the project that was imported into replit and click on src in the files of the project. Open the firebase folder and inside create a new file called firebaseConfig.js. In the new file, copy and paste the following code into the file.

//import firebase app for database initialization

import { initializeApp } from "firebase/app";

//import firebase database for database usage

import { getDatabase } from 'firebase/database'

//import firebase authentication

import { GoogleAuthProvider, getAuth, setPersistence, browserSessionPersistence } from "firebase/auth";

//Configure the firebase database

const firebaseConfig = {

//your firebase config information goes here

};

const app = initializeApp(firebaseConfig);//initialization of the firebase app

export const auth = getAuth(app);//initialization of the firebase authentication

//sets the browser persistence to session to only keep the users information during the current session

setPersistence(auth, browserSessionPersistence).then(() => {

console.log("Successfully set browser persistence");

}).catch((error) => {

console.log("Error setting persistence", error);

});

export const db = getDatabase(app);//initialize the firebase database

export const provider = new GoogleAuthProvider(app); //initialize the google auth provider for the google login features

After pasting the code, take the firebase config information from firebase and paste it into the firebaseConfig variable then save the file. After the file has been saved, configure the run button for replit to the command npm run start, or run the command in the provided console. Run the program, and once it has finished loading click on view in new tab on the project viewer and copy the url of the site in the new window. Return to the firebase console and under build click authentication. Click get started on the authentication page and click the option for Google authentication. Enable Google authentication and follow the onscreen instructions to create the project name and add the support email. Save the settings for the sign in information and navigate to the settings under the authentication tab. Under the settings menu click on authorized domains and click on add domain. Paste the copied url into the box and click add. The above steps ensure that the database is correctly set up, the authentication is valid through the site's url and that the site has installed all necessary packages to run.

## **Known Bugs**

A slight bug in the system is the rendering of the words in the word cloud. Due to the filtering being done for the word cloud, certain words are not rendered as they should be. Any words that may contain characters such as apostrophes within the words do not properly display this character. The filtering process strips any special characters out of these words, leading to some words not being displayed as they should be with the correct characters. This bug does not heavily affect the system, but it does change the display of some of the words, meaning that they do not look as they should.

The next bug in the system has to do with the login process. Even though it has been mostly fixed, an issue is still present for users who are attempting to login. On rare occasions when the user navigates to the page the site will automatically log them in without them clicking on the login button. This occurs because the user is still logged in somewhere across the browser with their Google account, so no issues occur with unauthorized users. This does not have the intended result however, because the idea is for users to have to log in every time they go to the site.

Another but in the system deals with the security of the site. Due to how the sites routing and security is done, it is possible for users to navigate to pages within the site without having to login through Google first. This is an issue because depending on what the user chooses to do while in this state, they can encounter uncaught runtime errors, effectively causing issues across the site. This happens when the user tries to access some data that requires them to have valid credentials, and the errors in that case are not being properly handled.

The navigation bar “Logout” link is of a different font size since it operates as an on-click function while the rest are links.

Some spacing of elements could use more consistency when switching between media-view.

Avatar selection in settings always begins on element one rather than the current-image of a user stored in local storage.

## Users may click the back arrow through different points on the site, taking them to the previous page. The site then has to reload information that it has already done. This is not an issue until the user logs out and then des this. Once the user has logged out and uses the back arrow, it will throw an uncaught runtime exception for them because it is trying to use information that does not exist for that user.

## **Future work**

### **Expanded Word Cloud Filtering**

The word cloud components on both the user’s homepage and the community page must be filtered to render correctly. Currently the word cloud utilizes a list of “stop-words” that are excluded from the cloud render. Casing and special characters are also excluded.

Future work on this component would include a more detailed stop-words list, the ability to couple together common strings of well used phrases, and the dynamic rendering of the words according to the prevalence of use. (ie if the word “Love” were used 10 times and the word “taco” was used 5 times then during rendering the word love would be of a larger font size than taco.

### **Theme Selection Component**

The color scheme used in the prototype application's design was purposefully grey scale with exceptions for elements that were to grab the focus of the user. A future design iteration would include a theme selection page accessible from the navigation pane. Users would be able to choose a global theme for the application that fit their preference. This would include font-family, background color, font-size and other pertinent elements.

This choice would be stored in the database and loaded when a user logged in to their account.

### **Expanded Administrator Functions**

The addition of greater use of administrative functions would be a top priority in future development. Along with the current ability for an administrator to delete accounts, the ability for a single super user to select which additional accounts could have admin functions would be available in the admin settings module inside of the application. At the time of prototype deployment, the admin functions are assigned and unassigned withing the database itself.

### **Cloud Masking Element**

The word cloud features are currently rendered in a circular form in order to imitate a “cloud” as is the theme of the application. Future development would see an SVG cloud image used as a mask to better shape the word cloud visually into the shape of a cloud like that of the Gratitude++ logo.

### **Audio/Video Content Expansion**

The initial vision of the Gratitude++ Journal Application was to capture user input in as many ways as possible. For the sake of initial production and limited time, the developers chose to limit input to text only. Future iterations of development would include the body of journal entries to be saved in video, audio, and image files.

### **Keeping user data**

The current way that user account information is handled when the account is deleted is to remove any trace of the account, which includes personal and community information. A future development of the project will see the information on the community page remaining on the site even after the account has been deleted. This will allow for more variety of entries and allows users to see posts that have been created that might interest them, even if the account no longer exists. Also, the capability for a user to download all their data.

## **References**

1. <https://legacy.reactjs.org/docs>
2. <https://nodejs.org/docs>
3. <https://firebase.google.com/docs>
4. <https://docs.github.com/en>
5. <https://docs.replit.com/>